2

OPY

Ada 9X Project Report

Shared Variables and Ada 9X Issues

January 1990

Office of the Under Secretary of Defense for Acquisition

Washington, D.C. 20301

REPORT DOCUMENTATION PAGE

Form Approved
OPM No. 0704-0188

1. AGENCY USE ONLY *(Leave Blank)*

2. REPORT DATE
January 1990

3. REPORT TYPE AND DATES COVERED
Final Report

4. TITLE AND SUBTITLE
Ada 9X Project Report, Shared Variables and Ada 9X Issues, January 1990

5. FUNDING NUMBERS
C = MDA-903-87D-0056

6. AUTHOR(S)
Robert B.K. Dewar
John B. Goodenough, editor

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

8. PERFORMING ORGANIZATION REPORT NUMBER
SEI-90-SR-1

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Ada Joint Program Office
1211 South Fern St., 3E113
The Pentagon
Washington, DC 20301-3080

Ada 9X Project Office
AF Armament Lab/FXG
Eglin AFB, Florida 32542-5434

10. SPONSORING/MONITORING AGE REPORT NUMBER

11. SUPPLEMENTARY NOTES
This report has been produced under the sponsorship of the Ada 9X Project Office. It is one in a series that addresses special issues relevant to the Ada revision effort.

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This report examines a number of issues surrounding the use and definition of shared variables in Ada in relation to possible Ada 9X requirements.

BEST AVAILABLE COPY

14. SUBJECT TERMS
Ada 9X, shared variables, pragma SHARED, ANSI/MIL-STD-1815A, Ada Joint Program Office, Ada 9X Project Office

15. NUMBER OF PAGES

. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT
UNCLASSIFIED

18. SECURITY CLASSIFICATION OF THIS PAGE

19. SECURITY CLASSIFICATION OF ABSTRACT
UL

20. LIMITATION OF ABSTRA

Ada 9X Project Report

Shared Variables and Ada 9X Issues

January 1990

Office of the Under Secretary of Defense for Acquisition

Washington, D.C. 20301

SEI-90-SR-1
January 1990

Shared Variables and Ada 9X Issues

Robert B. K. Dewar

New York University

This report has been produced under the sponsorship of the Ada 9X Project Office. It is one in a series that addresses special issues relevant to the Ada revision effort. John B. Goodenough, of the Software Engineering Institute, has served as the editor and coordinator for each report.

Accession For	
NTIS GRA&I	✓
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This work is sponsored by the U.S. Department of Defense. The views and conclusions contained in this document are solely those of the author(s) and should not be interpreted as representing official policies, either expressed or implied, of Carnegie Mellon University, the U.S. Air Force, the Department of Defense, or the U.S. Government.

Table of Contents

1. Shared Variables and Ada 9X Issues

This report examines a number of issues surrounding the use and definition of shared variables in Ada in relation to possible Ada 9X requirements.

1.1. Synchronous and Asynchronous Shared Variables

Before we begin an examination of the issues, it is important to define the distinction between two different types of shared variables, which in this document we call *synchronous* shared variables and *asynchronous* shared variables.

In Ada terminology, a variable is shared if it is accessed or modified by more than one task. For this purpose we assume (although it is not made absolutely clear in the reference manual) that the task declaring the variable is considered to access it by virtue of the declaration, so another definition is that a shared variable is one that is declared in one task, and accessed or modified in at least one other task.

In current Ada, programs are required, in the absence of pragma SHARED, to obey certain restrictions with respect to access to shared variables. These restrictions are stated in section 9.11 of the RM in a somewhat confused form (we will look at this confusion later), but the following is a summary of the intent of these restrictions.

- Between synchronization points (which primarily consist of rendezvous points, but also include task creation and termination), a shared variable must obey concurrent read, exclusive write (CREW) discipline. This means that if a task writes a shared variable, it owns the variable, and no other task can read or write the variable in that synchronization period. A program execution violating this rule is erroneous.
- Since it is hard to state the CREW restriction precisely, another characterization is that tasks are permitted to retain local copies of shared variables (in registers, local memories, caches etc.) and need only synchronize these local copies with the shared memory instance at rendezvous points. Any program whose effect depends on whether or not local copies are maintained is erroneous.

We will call shared variables meeting these requirements "synchronous shared variables." A typical example is a shared data structure where an agent task grants permission for access, so that a task wishing to modify the shared structure must request permission from the agent:

```
AGENT.GET_WRITE_ACCESS;         -- entry call to get access
...
(code to modify shared structure)
...
AGENT.RELEASE_WRITE_ACCESS;     -- entry call to release access
```

Assuming that all tasks accessing the shared structure make the required calls to the AGENT task, it is clear that access to the shared structure obeys CREW requirements, and there is no problem in the calling task working with a local copy of the shared structure, where the copy is taken immediately after the first entry call, and then stored back (if it has been modified) at the point of the second entry call.

Shared variables that do not meet these requirements are called "asynchronous shared" variables, reflecting the fact that any task can read or write the variable at any time. Local copies of such variables obviously cannot be maintained at any point, and there is a concern over whether or not access to an asynchronous shared variable is an atomic operation. In Ada, asynchronous shared variables must be marked with pragma SHARED.

Note: it is a source of continuing confusion that pragma SHARED does not correspond to the concept of shared variable, as defined in Steelman or the RM, but rather to a particular subset of shared variables, namely the asynchronous ones. This means for example that the Steelman requirement that shared variables be marked is not satisfied by the provision of pragma SHARED, since synchronous shared variables do not have to be marked in Ada.

1.2. Properties of Shared Variables

In discussing the issues surrounding the use of shared variables, we will use the terminology introduced in [5]:

1. Uniqueness

   The maintenance of local copies of variables may alter the semantics of programs in the presence of shared variables. There are two separable concerns in this regard:

   1. For variables that are *not* shared, it is important that the compiler be able to maintain local copies for optimization purposes.
   2. In programs making use of shared variables where local copies can affect the semantics, the programmer must be able to control, or at least know about, whether the compiler maintains local copies.

2. Independence

   In normal sequential Ada semantics, two separate variables (either separate scalar variables, or separate components of a single composite variable) are completely independent, in that modifying one of them does not affect the value of the other. Although there is no conceptual problem in maintaining this independence in the presence of tasking and shared variables, there are serious implementation problems.

3. Atomicity

   When a shared variable is accessed by two or more tasks more or less simultaneously, an important consideration in some cases is whether or not the update of the value is atomic. An atomic update guarantees that another task reading the value while it is being updated sees either the old value or the new value, but not some (possibly ill-defined) intermediate state.

1.3. Summary of the Issues to be Addressed

The following are the issues and possible requirements with respect to shared variables that are further discussed in this report:

- Use of shared variables for synchronization

  One intention of putting shared variables into the Steelman requirements and the Ada design was to enable efficient low level synchronization, avoiding the high level rendezvous mechanism. Does the current definition satisfy this requirement?

- Use of shared variables for memory mapped I/O

  A common practice is to associate variables with memory mapped I/O devices. Such variables are implicitly shared by the hardware. Is the current treatment of shared variables consistent with this usage?

- The independence problem for shared variables

  Variables which are notionally independent, e.g., elements of a single array, may in fact be dependent at the hardware level. For example, the bits of a packed Boolean array are independent variables, but at the hardware level, they may need to be accessed by words. Does the language definition deal with this problem correctly?

- Parallel and distributed requirements

  A clear requirement is that Ada 9X be compatible with parallel and distributed architectures. The general problem in such systems is distinguishing between local and global storage, when global storage is inefficient or non-existent in some systems. Efficient use of caching is also an important issue. Is the handling of shared variables consistent with these requirements?

- Shared variable definition

  Are the restrictions of 9.11, defining permitted access to synchronous shared variables, properly stated?

Clearly we want the answer to all these questions to be yes, and in all cases it can at least be argued that the answer is no for the current Ada definition. We will examine each issue in turn, identifying problems that must be solved during the language revision.

2. Use of Shared Variables for Synchronization

A typical example of the use of shared variables which was discussed during the design is the implementation of spin locks. The idea is that one task waits on a spin lock using code similar to:

```
    while BLOCKED loop null; end loop;
```

and another task releases the waiting task by simply executing the assignment statement:

```
    BLOCKED := FALSE;
```

The attraction of this type of synchronization is obvious. Assuming that the two tasks involved are running on separate processors with shared memory, it reduces the "synchronization" overhead to 1 or 2 instructions at the hardware level, and it is most unlikely that even aggressive optimizations of specialized task structures can achieve similarly efficient implementations of rendezvous synchronization styles.

In order for this approach to work correctly, it is necessary that the following rules be observed in generating code for accesses to BLOCKED:

- No local copies may be maintained.
- Every reference to BLOCKED must correspond to an actual load or store. The optimizer may not reorder references to BLOCKED or make assumptions about its value. For example, an optimizer cannot look at the loop and deduce that BLOCKED is a loop constant whose evaluation can be moved outside the loop.
- The update and access to BLOCKED must be atomic operations, so that consistent values are always obtained.

The intention is that pragma SHARED enforce these requirements. It is clear from the description that the first and third requirements are met, since they are explicitly stated in RM 9.11(9-11) (Appendix A contains all of section 9.11). However, it is far from clear that the second important requirement is stated. In RM 9.11(9), we have:

> *The pragma SHARED can be used to specify that every read or update of a variable is a synchronization point for that variable; that is, the above assumptions [CREW access] always hold for the given variable.*

Even a charitable reading has a hard time working out what is meant by this sentence. The "above assumptions" are stated in terms of accesses between synchronization points, and if every access is a synchronization point, then clearly there are no accesses between synchronization points. Historically, this section underwent a number of radical changes in descriptional sty' . At one point there was a concept of a "phantom task" associated with the variable, with accesses representing rendezvous actions with this task. This phantom task concept probably can be stated in a way that enforces the second requirement above, but what has eventually been left in the RM is probably quite inadequate.

The optimization section, 11.6, needs to be checked carefully to make sure that it does not permit derailing of code involving pragma SHARED variables. Consider the following examples:

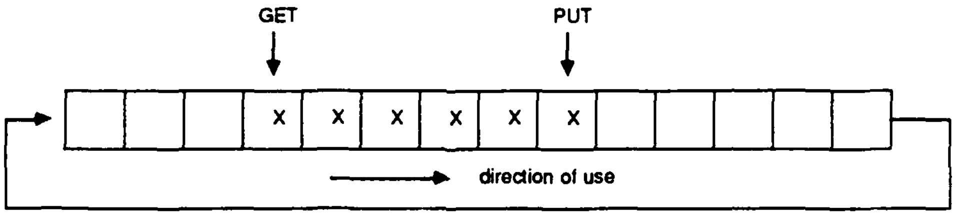
```
A := 2;            -- Can this assignment be suppressed?
A := 1;

B := A;            -- Can this reference to A be suppressed?
B := A;

loop
   C := A + 1;     -- Can A+1 be pulled out of loop if A is unchanged?
   ...
end loop;

B := A1;           -- Can references to A1 and A2 be interchanged?
C := A2;
```

For normal variables A, A1, and A2, the answer to all these questions is yes. However, if A, A1, and A2 are pragma SHARED, the answer should probably be no. The current rules in section 11.6 *do* prohibit interchanging assignment statements, but this rule is probably inappropriately severe for normal non-shared variables. However, if this rule is relaxed for Ada 9X in some form, it is important to remember that the rule *is* appropriate for pragma SHARED variables.

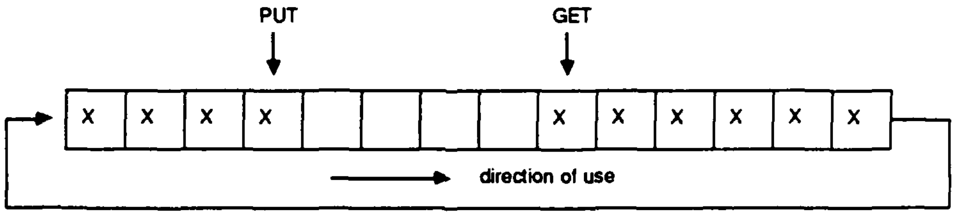
Anyway, it seems well within the bounds of reasonable interpretation of the current RM to assume that the language is intended to guarantee that the suggested spin lock implementation is required to work, providing that the implementation accepts the pragma SHARED. Indeed, the spin lock example was often used during the design phase to justify the inclusion of shared variables in the language. Since there is no hardware for which it is reasonable to claim that it is impractical to represent a BOOLEAN variable in a form permitting atomic updates, it is hard to see that any implementation could legitimately reject this pragma.

For a case where the current model of pragma SHARED seems clearly inadequate, consider the case of a circular, or bounded, buffer implementation with a single producer and a single consumer. This is a familiar scheme which is frequently used in operating systems. For example the communication between the keyboard interrupt handler and the keyboard read routines on PC's uses such a scheme.

The idea is to have a vector of data items, which at any particular moment is partially full:



Here the Xs represent data items currently stored, and as indicated, the vector is addressed in a circular manner, so that the data can wrap around the end:



The producer places data in the buffer using the PUT pointer, being careful to put the data in place BEFORE incrementing PUT. The consumer removes data using the GET pointer, being careful to remove the data before incrementing GET. If this order of operations is followed then there is no need for any other synchronization, and again we achieve task synchronization at the single memory reference efficiency level.

The following is a sketch of an attempt to program this approach in Ada:

```
-- The shared global data items:

   BUFFER   : array (INDEX) of DATA_ITEM;
   GET, PUT : INDEX;

-- The code in the producer task (assume + is mod length of array):

   while PUT + 1 = GET loop          -- wait for available slot
       null;
   end loop;

   BUFFER (PUT + 1) := DATA;         -- put data in
   PUT := PUT + 1;                   -- increment put pointer

-- The code in the consumer task:

   while PUT = GET loop              -- wait for data to appear
       null;
   end loop;

   DATA := BUFFER (GET);             -- get data
   GET := GET + 1;                   -- increment get pointer
```

If this works, it is certainly a highly efficient approach for synchronizing the two tasks. Does this approach work in Ada? Well first of all, it is clear that PUT and GET must be pragma SHARED, since the key to correct operation is that these variables are asynchronous shared variables with atomic update and access. One cannot necessarily assume that all implementations will accept the pragma SHARED declarations for PUT and GET. For example, on a system containing two 8088 8-bit processors, where type INDEX requires 16 bits, it may be impractical to support pragma SHARED for INDEX. In such a case, the compiler should reject the pragma SHARED declarations (i.e., issue a warning that they are ignored), and a programmer is then informed that there is no alternative but to resort to explicit synchronization, or perhaps type INDEX can be fitted in 8 bits in this case.

However, on almost all implementations, it is likely that PUT and GET can be successfully declared using pragma SHARED, giving them the required characteristics. With these declarations, the above code will in fact work on most current implementations of Ada. However, it is still erroneous, and it is entirely possible that future implementations of Ada will *not* implement this code as intended.

The point at which execution becomes erroneous is on the access to elements of BUFFER. These elements are shared variables which are accessed asynchronously. The two tasks have indeed synchronized using PUT and GET, but this is not synchronization in the official sense. Consequently it is perfectly permissible for the two tasks to work with local copies of BUFFER. In practice, on most machines with shared memory, there will be only one copy of BUFFER. However, in distributed systems with no shared memory, or in multi-processing systems with non-coordinated caches (the cache issue is discussed in more detail on page 20), it is quite reasonable that the tasks work with local copies of BUFFER, which of course renders the entire approach invalid.

The current design of pragma SHARED allows it to be applied only to scalar and access variables with simple names, so there is no way to designate the elements of BUFFER as SHARED. Furthermore, extending pragma SHARED so that it can apply to elements of a composite object is still not right. The elements of BUFFER are indeed asynchronously shared, in that they are accessed without explicit synchronization, and hence no local copies can be maintained. However, pragma SHARED in its current form also requires that the access be atomic. This is *not* a requirement for the elements of BUFFER. Indeed it is quite possible that the elements of BUFFER are relatively large objects for which atomic access is impossible.

BUFFER is thus an example of a type of object not envisioned in the current Ada design, for which local copies must be suppressed, but atomic updates and references are not required. The reason is that the logic of the program provides the necessary logical synchronization to ensure that a given element of BUFFER can never be read and written at the same time by separate tasks. Of course there is no way syntactically of enforcing this requirement, it is a (possibly arbitrarily complicated) consequence of the logic of the program that this restriction is met.

What is needed is some way of enforcing the requirements of no local copies, and also the requirement that every reference correspond to a load and every assignment correspond to a store, *without* also requiring atomic access. This could for example be supplied by extending the functionality of pragma SHARED:

```
    pragma SHARED (variable_simple_name, ATOMIC => YES);
    pragma SHARED (variable_simple_name, ATOMIC => NO);
```

where variable_simple_name is now permitted to name a composite object (in which case the condition applies to the lowest level components if ATOMIC is set to YES). The default for ATOMIC could be YES to maintain compatibility with the current language. This is not necessarily an attractive syntax, nor is it intended to be. It simply illustrates the kind of approach that is needed to give the required functionality. Note that in the case of NO being specified for ATOMIC, the definition would have to state that simultaneous access and update was erroneous. Informally this is a straightforward idea, however, formalizing it is tricky because of the reliance on the notion of time.

The bounded buffer is simply an example of a whole class of algorithms which access shared memory in this kind of manner. Another example is concurrent garbage collection algorithms. Indeed an article in TOPLAS describing such an algorithm used Ada as the descriptive language, and the Ada was, as the above discussion indicates, irreparably erroneous. The whole point in concurrent garbage collection algorithms is to use shared memory access for synchronization. It is certainly unacceptable to make an explicit synchronization call for every access to heap memory.

At least one implementation has recognized this problem by providing a pragma with the desired semantics. The pragma VOLATILE in DEC Ada is precisely a guarantee that no local copies are kept and that no optimizations are applied. It does not specifically speak to the issue of atomicity, but in practice small objects up to 32 bits are treated atomically, and large objects are treated non-atomically, so, at least in the context of a particular architecture, the full range of possibilities is provided (at least in early versions of this compiler, pragma SHARED was completely ignored).

A further significant problem in this area is that the bounded buffer example will undoubtedly work in most current Ada implementations because typical mono-processor Ada implementations will never take local copies of arrays. However, on multi-processors with uncoordinated caches, local copies of all data is made automatically as a result of caching actions, and programs of this kind will not work. For a further discussion of this point, see [2].

3. Use of Shared Variables for Memory Mapped I/O

A common practice is to associate variables with memory mapped I/O devices, using address clauses. The following is an example:

```
type IO_REC_TYPE is
    START_ADDRESS : INTEGER;
    LENGTH        : INTEGER;
    OPERATION     : OPERATION_TYPE;
    RESET         : BOOLEAN;
end record;

IO_REC : IO_REC_TYPE;

for IO_REC use at 16#4000_FF00#;

...
...

IO_REC.START_ADDRESS := JUNK'ADDRESS;
IO_REC.LENGTH := 32;
IO_REC.OPERATION_TYPE := READ;
```

In this example, the hardware structure controls the input/output operations, and a store in the OPERATION field triggers the operation. It is also possible to design hardware where simply reading a memory location triggers operations. and in this example we imagine that simply reading the RESET field will cause the current operation to be terminated.

There are number of problems associated with this approach, some of which have to do with the shared variable issue.

First, the representation clause giving the address is implementation dependent. This is not in itself a problem, since of course this kind of code is not expected to port to other hardware environments. However, there may still be problems in writing the necessary representation clause. The Ada RM does not make clear whether the type SYSTEM.ADDRESS is to be treated as a virtual or physical address. Very often implementations naturally make this type be the virtual address in a system where the two address spaces are distinct, so that A'ADDRESS is related to or identical with a pointer to A (a lot of interface code informally depends on the unjustified assumption of such equivalence). On the other hand, a memory mapped I/O device presumably is at some particular physical address. The RM does give permission (RM 13.5(3)) for "fudging" the interpretation of addresses:

> *The conventions that define the interpretation of a value of the type ADDRESS as an address, as an interrupt level, or whatever it may be, are implementation dependent.*

Nevertheless it is not clear what constitute reasonable conventions in such a situation. Additionally, addresses often correspond to unsigned variants of the largest supported integer type, and consequently are tricky to specify, given Ada's lack of support for unsigned integers.

The reason that we address this issue, even though it has nothing to do with shared variables, is that difficulties of this type have led some Ada language lawyers to decry the use of address clauses for the purposes of memory mapped I/O. It is indeed the case that this usage is fraught with problems. Nevertheless, it is clear from practice that users strongly desire to use this approach, and regard the possibility of writing this kind of code as a strength of Ada *if* it works. Thus it may be desirable for Ada 9X to accommodate this usage.

Once one permits this kind of notation and approach, the shared variable issues arise because even if only a single Ada task touches the I/O variable, it is being referenced independently by the hardware and is thus logically a shared variable.

- No local copies may be maintained.
- Every reference must correspond to an actual load or store. The optimizer may not reorder references, or make assumptions about its value.
- The update and access to fields must be atomic operations.

The last requirement reflects the fact that the exact pattern of touching the relevant variables affects the operation of the device. For example, it is normally acceptable though inefficient to update a 16-bit field with two separate 8-bit store instructions, but in the case of memory mapped I/O devices, the effect might be radically different.

The above requirements are essentially identical to those which seem to be required for pragma SHARED. Actually they are a little more strenuous. Consider a sequence of accesses:

```
X := SHARED_VARIABLE;
X := SHARED_VARIABLE;
X := SHARED_VARIABLE;
X := SHARED_VARIABLE;
```

If SHARED_VARIABLE is being used only for inter-task synchronization purposes, it *would* in fact be valid to skip the first three references, since they could not affect the logic of the program. However with a memory mapped I/O device, every reference may actually trigger some operation, so it would be improper to suppress the accesses.

For the synchronization case, it would have been acceptable to formulate a more relaxed rule permitting the optimizer to remove dead references, but this would have introduced unnecessary complexity (it is unlikely that real programs contain such redundant references), and anyway, as we see in examining the memory mapped I/O case, we need the stricter rule in this situation in any case.

In terms of the existing language, we again run into the problem that the current definition of pragma SHARED is too restricted, in that we may need to deal with components of a composite object. It is also the case, as this example illustrates, that we may need to designate *atomic* access for some elements of a record, and not for others. This further complicates the design of an appropriate extended version of pragma SHARED.

There is one additional problem in the case of memory mapped input/output. Consider the case of a 2-byte variable on a machine with both 2-byte and 4-byte loads. For the normal case of asynchronous shared variables, even those for which atomic access is required, it is permissible to generate a 4-byte load to access the required 2-bytes. This is not a far-fetched possibility. At least one current 386 Ada compiler will generate code of the form:

```
V16_A := V16_B;

MOV  EAX,DWORD PTR V16_B;
MOV  V16_A,AX
```

since the load of a 32-bit word is more efficient than the load of a 16-bit word on the 386. However, this code is potentially seriously incorrect in the case of memory mapped I/O devices, since the access to the two extra bytes may trigger some entirely unexpected event.

A similar situation can arise in the following case:

```
type P is array (0..31) of BOOLEAN;
pragma PACK (P);

IO_DEVICE : P;
for IO_DEVICE use at ...;

OPCODE renames IO_DEVICE (N..M);

OPCODE := READ;
```

In this sequence of code, particularly if N and M are dynamically computed, it may well be the case that the compiler generates rather general access code which accesses bits outside the bounds of IO_DEVICE. With at least one current compiler, this problem arises even for a simple renaming of one of the fields of a record:

```
OP renames IO_REC.OPERATION_TYPE;
...
OP := READ;
```

The compiler stores a dynamic bit address for OP, and the access to OP then uses generalized code that may load bytes outside the bounds of IO_DEVICE.

This is certainly a very tricky problem to approach at the language level. From one point of view, it is tempting to simply write all this off as an implementation dependent consideration. However, both users and implementors need at least some guidance on what should or should not be expected to work when address clauses are used for memory mapped I/O devices.

The problem is essentially that the programmer needs to be able to control the exact sequence of machine instructions used to access the memory mapped location. This seems incompatible with the general semantic view of Ada, which is independent of the sequence of generated machine instructions. It is conceivable that this problem could be addressed by taking another look at machine code insertions together with pragma INLINE applied to such insertions.

Note that the pragma VOLATILE of DEC again solves much of this problem, though it does not specifically address the dynamic bit reference issue. Since the Vax architecture has bit field extraction instructions, the possibility of referencing bits outside the target field probably does not arise on this machine.

4. The Independence Problem for Shared Variables

The current rules in 9.11 refer only to scalar and access variables in discussing the limitations on access. This was in fact a last minute change. The printer's proof still omitted this restriction. Without this restriction, separate tasks could not access separate parts of a composite object, since the access to the object as a whole would violate the shared variable assumptions.

The last minute change removes this restriction, and in so doing enables the programming of otherwise impossible tasks, such as the following:

- Quicksort implemented with separate tasks working on separate sections of the array.
- Numerical operations on two dimensional arrays, with different tasks working on different rows.
- Graphical display management with different tasks modifying different sections of a shared bit mapped display area.

In all these cases, there are no shared variables according to the Ada definition, since separate tasks are working on separate components of a single composite object. No *single* component is accessed by more than one task without proper synchronization, so the restrictions on the use of shared variables are not violated.

Although this makes abstract sense, there are two important situations in which the assumption that individual components of a composite object are independently accessible breaks down.

The first arises in the case where representation clauses are used to lay out bit fields in a record, or pragma PACKED is used to compact elements of an array. In this case, the unit of access at the hardware level may encompass more than one component. Suppose we have:

```
type WORD is array (0..31) of BOOLEAN;
pragma PACKED (WORD);
W : WORD;
```

Now suppose we have two tasks which, "at the same time," execute the following assignments:

```
W(3) := TRUE;                W(4) := TRUE;
```

The semantics require that these two assignments be independent, but on a typical RISC machine, the code sequences generated will be:

```
R1 <-- W;                 R2 <-- W;
R1 <-- R1 or 2#1000#      R2 <-- R2 or 2#10000#;
W  <-- R1;                W  <-- R2;
```

These sequences of code are certainly *not* independent, since if they are executed in an interleaved manner:

```
R1 <-- W;
R2 <-- W;
R1 <-- R1 or 2#1000#;
R2 <-- R2 or 2#10000#;
W  <-- R1;
W  <-- R2;
```

the effect of one of the assignments is completely lost, obviously violating the independence principle. The only way to avoid this would be to generate explicit synchronization code, which is clearly not the intention (synchronization code is expected only for a rendezvous).

If the hardware lacks the capability of bit access at the memory level, it may simply be impractical to generate reasonable code for twiddling one bit without disturbing other bits. What is a compiler to do in this circumstance? AI-00004 proposes to solve this problem by extending the wording of 9.11(4, 5) to include variables for which representation pragmas or pragma PACKED is present as well as scalar and access variables. However, this AI has never been approved.

This is a very important problem. A significant number of implementations, including all those for RISC chips, are simply wrong at the current time, and there is no way that they can be corrected. It certainly must be addressed for Ada 9X, and should indeed be addressed for Ada today. Note that the more recent AI-00555 requires that pragma PACKED be implemented as close packing for type BOOLEAN, further aggravating the problem.

As usual, the subunit case is particularly embarrassing, because it creates a situation in which a normal non-shared variable is not known to be non-shared at compile time, because there could be tasks within the subunit causing problems in simultaneous access.

A potentially more troublesome version of this problem arises in the case of parallel machines with caches. This is discussed in the following section.

5. Parallel and Distributed Requirements

The following possibilities characterize various parallel and distributed architectures:

1. Memory may be divided into local and global sections with various access possibilities, such as:
   - Local memory may be directly accessed only by its own processor
   - Accesses to local memory are more efficient than to global memory
   - Even if local memory can be accessed by other processors, such access patterns may lead to memory bottlenecks.
2. There may be no global memory at all, resulting in considerable inefficiency if a processor must access memory which is not local to the processor.
3. The individual processors may have separate caches. Particularly in highly parallel systems, these caches are likely to be non-coherent, which means that if the same data exists in several caches, it is up to the program to ensure that it is consistent — the hardware does not automatically flush or update the cache if the corresponding memory location is modified by some other processor.

It is important that Ada be compatible with all these architectural possibilities. By compatible, we do not mean that any possible Ada program should be able to run efficiently on all possible architectures. This would clearly be an unrealistic requirement, since for example it is clear that a program in which multiple tasks have heavy patterns of access to shared global memory cannot possibly run efficiently on a system with no shared global memory if tasks are placed on separate processors. Rather the requirement is that it be possible to write Ada programs in a manner that is reasonably natural to the architecture at hand, and expect that the compiler will map the program to the hardware in an efficient manner.

5.1. Classification of Variables – Marking Shared Variables

In a system with local and global memory, where either for efficiency or functional reasons it is important to locate variables in the appropriate section of memory, it is clearly important for the compiler to be able to identify the usage of variables.

Variables that are not shared can clearly be placed in the local memory of the processor which is executing the task which accesses the variable. Shared variables on the other hand, must either be located in global memory, or, in a machine with no global memory, accesses to these shared variables will involve transmitting communication messages to an agent on some particular processor who owns the variable.

Particularly on fully distributed, loosely coupled systems with no global memory, this type of communication may be expensive, so one can expect that on such systems, heavy use will be made of the permission to keep local copies of synchronous shared variables, even possibly for large composite objects. Ada, via pragma SHARED, clearly indicates variables for which local copies may *not* be made, which is of course important since inhibiting local copies can be very expensive in such architectures.

Unfortunately, Ada as it stands now does *not* distinguish clearly between non-shared and synchronous shared variables. Consider the following procedure definition:

```
procedure X is
  Q : array (1..16) of INTEGER;
  procedure SEP is separate;
begin
  ... accesses to Q
end;
```

The code corresponding to accesses to Q may be quite different depending on whether Q is shared or not, because local and global accesses may involve very different mechanisms. In the extreme case, as we noted above, accesses to global memory for shared variables may involve the transmission of communication messages.

Unfortunately, at the time that procedure X is compiled, it is not possible to tell whether or not Q is shared. Probably it is not shared, since even if X is called by several tasks, each call has its own instance of Q, and the instance is probably local to the calling task. The exception arises if the body of SEP contains tasks which reference Q, in which case it *is* shared after all. There is of course no way to know this at the time that X is compiled. As Susan Flynn has shown [3], it is possible at bind time to do a reasonably good job of detecting shared variables, but (a) it is still not perfect and has to make some pessimistic assumptions, and (b) it is generally impractical to defer code generation for the entire program until bind time.

In the absence of a bind time analysis, the compiler is forced to make worst case assumptions, and in particular the use of subunits (which are very widely used in some Ada coding styles), results in huge numbers of non-shared variables being considered as potentially shared.

A second issue involves library packages. Consider the following package body definition:

```
package body Q is
   X : INTEGER;
   ...
end Q;
```

Even if one can be sure at compile time that there are no tasks within Q which reference X, it cannot be determined whether or not X is shared. If multiple tasks reference Q, it is quite possible that X is shared. On the other hand, it may be the case that the logic of the program is such that only one task ever references X, and obviously this cannot be determined by examination of the program by the compiler.

Steelman had a requirement that all shared variables be marked. Although it may seem at first glance that pragma SHARED meets this requirement, this first glance is misinformed. As we noted earlier, the name SHARED in the pragma is confusing since it refers only to asynchronous shared variables. The Steelman requirement applies to all shared variables, including synchronous shared variables.

The failure of Ada to meet this requirement is exactly what causes trouble in the parallel and distributed case where the identification of shared variables becomes important. An extreme case arises in a system with no shared memory at all. A programmer writing an Ada program which is intended to run efficiently on such an architecture may be careful to organize the program so that all communication of data between tasks is via entry parameters. This model is an excellent match for architectures like hypercubes or networks of transputers, which are optimized for transmission of messages, but are not expected to be able to address a common global memory. A programmer proceeding in this manner would expect that the Ada compiler would map the program efficiently to the transputer network, but unfortunately it is likely that the compiler would not be able to tell at compile time that no variables were shared, with the result that it would generate inefficient code preparing for the non-existent eventuality that some of the variables would be referenced directly from other tasks.

The only solution to this problem seems to be to introduce pragmas or some other mechanism to allow a programmer to indicate which variables are shared and which are non-shared. One problem with such an approach is that it is potentially not upwards-compatible with 1815A. Obviously there are many Ada programs written today which do *not* mark the shared variables, since there is no way to do so in 1815A. We could require non-shared variables to be marked, but since virtually all local variables of procedures are non-shared, this would be painful in the extreme.

The following approach has been adopted by the NYU compiler for the IBM RP3. This machine has 512 independent processors with both local and global memory. It is important from an efficiency point of view to place variables in the appropriate memory area. The following pragmas are introduced:

```
pragma SHARED_VARIABLES_MARKED (DEFAULT => memory_type);
pragma MEMORY (variable_name, memory_type);
```

The SHARED_VARIABLES_MARKED pragma provides a default memory allocation type, and promises that all variables not subsequently marked with pragma MEMORY can be assumed to be of this type. The MEMORY pragma can then be used to mark variables which are not of the default type. If no SHARED_VARIABLES_MARKED pragma appears, the default is to assume synchronous shared variables living in global memory. Note that if a variable is assigned to local memory and is then shared, the result is formally erroneous, and in practice the resulting code will not only be inefficient, but due to caching issues discussed in the next section, will actually malfunction.

This approach is reasonably convenient and is completely upwards compatible. For the RP3 project, which is an experimental research project into the optimal use of memory allocation on an architecture of this type, there are a large number of possibilities for memory_type, but in the basic Ada definition, the only requirement is to distinguish non-shared from shared. The further distinction between synchronous shared and non-synchronous shared is made by pragma SHARED. Of course it is a pity that 1815A has absconded with this pragma name, which might better be used for marking shared variables than for designating them as asynchronous. The design of a pleasant and yet compatible set of pragmas to control this is not an easy task.

5.2. The Effect of Caches

Caches represent an automatic method for keeping local copies of frequently used variables.

The important reason for distinguishing synchronous shared variables as a special case is that an implementation may keep local copies of such variables, provided that these local copies are synchronized with the shared copy at synchronization points. The use of local copies can improve code performance in two common situations:

1. Variables may be kept in processor registers, avoiding the need to reload the values from memory on every access. Particularly when a variable can be held in a register over a loop, significant gains are possible. This is an absolutely standard kind of compiler optimization, normally performed in all compilers for sequential languages. To meet the additional requirements for synchronous shared variables, any such variables held in registers whose values have been modified must be stored in memory at synchronization points.
2. In the case of multiple processors each having independent caches (i.e., caches which can contain potentially incoherent results), caching is effectively the same as keeping local copies. Such local copying is often more aggressive than the use of registers, since local copies are taken of elements of composite structures in cases where not even very aggressive compilers would place the values in registers. For example, consider the code fragment:

   ```
   X := A(J);
   Y := A(K);
   ```

   If the compiler knows nothing about the values of J and K, it cannot usefully optimize this sequence. However, a cache will take a local copy of A(J), and dynamically, if J = K, reuse this local copy for the reference to A(K).

It is of course always valid to keep local copies of non-shared variables, so these variables can be cached freely. Furthermore it is *not* necessary to flush these variables from the caches at synchronization points. In the case where a cache can contain both non-shared and synchronous shared variables, it may or may not be practical to partially flush the cache. However, there are certainly situations in which it *is* practical. For example on the RP3, memory can be marked as cacheable, non-cacheable, or marked-cacheable. Both cacheable and marked-cacheable data is cached, but there is a specialized flush instruction which flushes only the marked data. On this machine, marking non-shared data in Ada as cacheable, and synchronous shared data as marked-cacheable, and then issuing a flush-marked instruction at rendezvous points corresponds exactly to the required Ada semantics.

Generally we cannot keep local copies of asynchronous shared variables in registers or caches (such variables need to be marked non-cacheable, which is a typical hardware possibility in all cache controlled systems). The one exception to this rule arises if only one task writes the variable. In this case the distinguished task that writes the variable may keep a local copy of the variable provided that the shared copy is also updated whenever the value is modified. This behavior is typified by that of a "write through" cache, which always writes the value through to main storage. Although we note this possibility, it is unlikely that a compiler could detect such a specialized situation automatically, although again it would be perfectly reasonable to have pragmas advising the compiler of this situation.

In making effective use of the cache it is again important to distinguish between non-shared and shared variables, and as we discussed in the previous section, this is not easily possible in Ada.

It might seem that we can simply cache everything except asynchronous shared variables marked by pragma SHARED. The only expense of such an approach appears to be that at synchronization points we may be forced to flush variables which are in fact non-shared, but we don't know this at compile time. However, for typical cache designs, there is another factor which again makes it imperative to know which variables are or are not shared. Consider a procedure declaration:

```
procedure X is
   A, B, C, D : INTEGER;
   procedure Y is separate;
begin
   ...
end;
```

Now we know that A, B, C, and D are not asynchronous shared, because no pragma SHARED appears for these variables. So they can potentially be cached. However, in a typical cache design, the width of a particular cache line is wider than an INTEGER (for example on the 68030, an INTEGER is presumably four bytes, but a cache line is 16 bytes).

If A, B, C, and D are non-shared, as is almost certainly the actual case, then they can share a cache line. If however, they are synchronous shared and accessed by different tasks, then although they can be cached, they must be placed in separate cache lines. This is because a statement like:

```
A := A + 1;
```

can actually result in reading and writing B, C, and D since they are part of the same cache line. From the point of view of the task executing the assignment, this write is harmless since the values of B, C, and D are not modified, but if in between the read and the write some other task modifies B, then the modification is lost, and the result is definitely incorrect.

More subtle cases of this issue of cache coherency arise. Consider the following case:

```
procedure ARRAY_MUCK is
   X : array (1..N, 1..2) of INTEGER;
   task body T1 is
     -- mucks with column 1 of X;
   end T1;
   task body T2 is
     -- mucks with column 2 of X;
   end T1;
   ...
end ARRAY_MUCK;
```

The elements of X must be arranged so that there is no single cache line containing elements from both column 1 and column 2. This will occur naturally if N is a multiple of the number of INTEGER's which fit in a cache line, but otherwise the structure of X must be modified.

Much more complicated examples can be drawn in which the way that a composite object must be laid out on cache lines depends in very delicate ways on the exact pattern of logic in the program. One approach is to develop a complex pragma language to explain the required layout to the compiler. In the absence of this, it is important for the implementor to understand what is and what is not valid, and for the programmer to understand the possibilities of inefficient translation which may result (e.g., a large array forced into a storage organization with one INTEGER for each 16 bytes).

5.3. Implicit Shared Variables

As we have discussed, in fully distributed implementations with no global memory, any use of shared variables is potentially embarassing from an efficiency point of view. There are a number of instances in which the Ada semantics tends to imply (and certainly results in practice) in the creation of shared data that is not explicitly declared by the programmer. Two examples are the bounds values for dynamically created subtypes, which are written by the task defining the type, and read by any other task referencing the type. Another case arises with ABE (access before elaboration) checking on procedure calls. Logically there is a BOOLEAN value associated with the procedure, which is set when the procedure is elaborated, and checked on a call. This flag is shared by all tasks using the procedure. In meeting the reuquirement for facilitating implementation of Ada programs on fully distributed systems, the Ada 9X design should take into consideration the status of implied shared variables of this nature.

6. Shared Variable Definition

This section addresses some of the technical difficulties in the current definition of shared variables in the Reference Manual. This is one of the more obscure areas of the manual, and one would hope that it will be cleaned up for Ada 9X.

An initial point to discuss here is the basic definition of a shared variable. In section 9.11(2), we have the comment:

... a shared variable (that is, a variable accessible by both [tasks])

What exactly does accessible mean? It could be taken to mean something like "potentially accessed," i.e., visible according to the visibility rules. Under this interpretation many variables that are in fact only accessed by a single task would be considered shared:

```
procedure Q is
   G : INTEGER;
   task X is ... no references to G ... end X;
   task body X is ... no references to G ... end X;
begin
   ... code using G
end Q;
```

Is it really sensible to regard G as shared in this example? G is certainly accessible from X, it just doesn't happen to be accessed. It is however probably more useful to regard X as not being a shared variable, and the definitional language should make this clear.

Section 9.11 of the RM attempts to define the restrictions corresponding to what we called synchronous shared variable access, but the attempt is flawed in a number of respects, which we will summarize here. For a fuller treatment of these issues, see [5].

The restrictions are currently stated in 9.11 (3-6) as follows:

For the actions performed by a program that uses shared variables, the following assumptions can always be made:

- *If between two synchronization points of a task, this task reads a shared variable whose type is a scalar or access type, then the variable is not updated by any other task at any time between these two points.*
- *If between two synchronization points of a task, this task updates a shared*

variable whose type is a scalar or access type, then the variable is neither read nor updated by any other task at any time between these two points.

The execution of the program is erroneous if any of these assumptions is violated.

The addition of the restriction "scalar or access type" is specifically intended to allow separate tasks to access separate components of a composite object. A formal question arises of whether updating or reading a composite object constitutes updating or reading its components. Probably answering this question from a purely formal point of view requires some careful exegesis, and it is likely that the answer might not be consistent — it is certainly not obvious what the answer is. However, for the purpose of the above quoted paragraphs, it is clear that the intention is that reading or updating a composite object *does indeed* involve reading or updating the components (otherwise simply making an object into a one component record and always using the record object would circumvent the intention of the rules). The wording should be cleared up here in Ada 9X.

A second problem arises with the definition of synchronization point. The definition in 9.11(2) is as follows:

Two tasks are synchronized at the start and end of their rendezvous. At the start and at the end of its activation, a task is synchronized with the task that causes this activation. A task that has completed its execution is synchronized with any other task.

Particularly in the last case, it is not clear how this relates to the idea of synchronization points. The intention is probably something like:

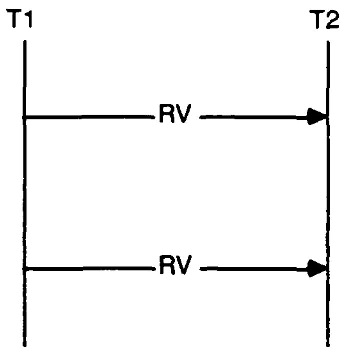
Synchronization points for a task include the following:

- The start and end of the activation of the task
- Elaboration of an allocator which activates a task
- Elaboration of a BEGIN which activates one or more tasks
- Elaboration of an ACCEPT statement
- Elaboration of an ENTRY call
- The end of a rendezvous (the END of an ACCEPT)
- Task termination

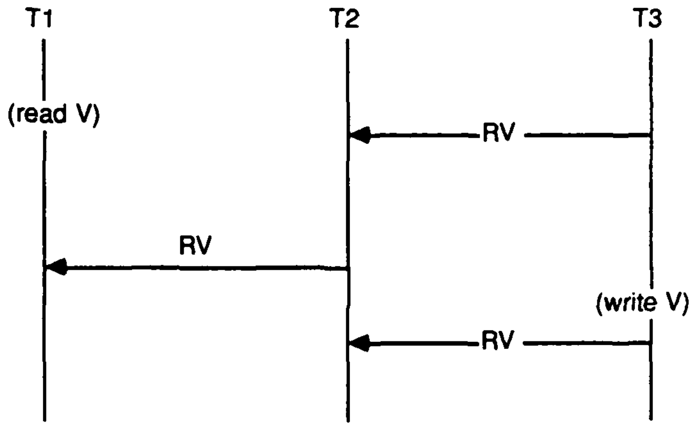
Certainly the definition needs to be in terms of events at a particular point in their execution.

A final problem with the language of paragraphs 3-6 is that the rules are phrased in terms of a situation arising between two rendezvous points, implying that the program execution does not become erroneous until the second synchronization point (note that the word "successive" is implied in the wording, though not explicitly present). This is clearly not intended, and leaves the status of programs which never terminate, and tasks which therefore never reach the second synchronization point, unclear.

Furthermore, the definition in the reference manual seems to be in terms of time, since "between" can only have the interpretation of time. For the simple case of two tasks synchronizing with one another:

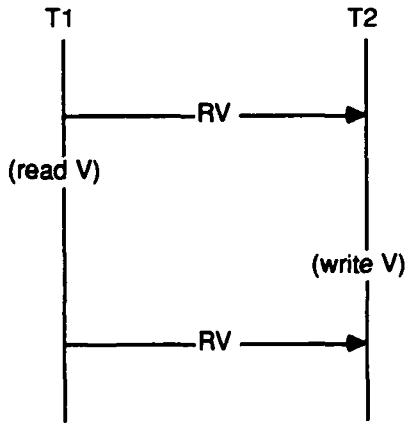


there is no difficulty since the "times" are determined by the rendezvous points which are clearly connected with specific points in the text. However, consider the case of three tasks with the following synchronization pattern:

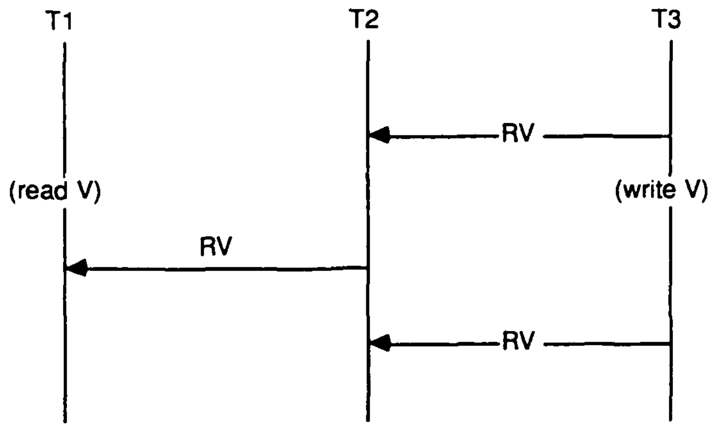


Now with respect to the access to V, the RM is phrased with respect to the period between the two RV's of T3, with the result that the program is erroneous if and only if the read of V in T1, which is at that time executing quite independently of T2 and T3, occurs before or after the first RV of T3.

This kind of time dependence almost certainly is not intended. To see this, consider the simple case with two tasks again:



Even though the write of V occurs after the read in time, so that in practice there would be no implementation difficulty, the RM clearly says that this program execution is erroneous, the intention being that in other perfectly valid execution interleavings, the read might occur at the same time or after the write. Similarly in our three task example, the execution should be considered erroneous because an essentially equivalent execution pattern:



causes a simultaneous read and write to occur. Thus the idea is that a program must be written to avoid the possibility of a simultaneous read and write, and a program not so written has an erroneous execution, even if by luck it does not encounter the embarrassing case.

To find the correct statement of the CREW condition, we first give an informal definition, which gives a better intuitive understanding of the rule from a programmer's point of view:

- Within a single synchronization period, if a task writes a variable, it can be sure that no other task can read the variable at the same time, i.e., that it has exclusive ownership of the variable for this period.
- Within a single synchronization period, if a task reads a variable, it can be sure that no other task can write the variable at the same time.

Here a synchronization period is either the period between two successive synchronization points, or it is the period starting with the last synchronization period of a task (up to the current point in execution). Actually the second condition is redundant since it is implied by the first, but it is helpful to state both cases, since the first relates to the EW (exclusive write) of CREW and the second to the CR (concurrent read).

The problem is to formalize this notion. One possible approach might be based on the model in the EEC Formal Ada definition. This definition has attempted to provide a model in which each variable is notionally tagged with its synchronization history, and an execution becomes erroneous at the point where an improper access occurs. Another source of ideas is work on parallel debugging by Edith Schonberg and Emanuel Ichbiah [4], which attempts to dynamically identify improper access patterns in executing Ada programs. Bryan's work [1], which presents an alternative formal model of task interaction, may also be useful in reformulating these rules.

Another source of inspiration may be found in Bryan's work [1], which presents another possible formal model of task interaction.

7. Possible Ada 9X Requirements

The following are examples of possible requirements for Ada 9X. They are intended to give an idea of the appropriate level for constructing requirements with respect to shared variables. These are rather high-level requirements, but it is hard to be any more specific without suggesting specific solutions.

- The possible presence of shared variables should not have any negative consequences on the efficiency of code which does not involve shared variables.
- The needs of distributed applications, where shared variables can cause severe problems if there is no shared memory, must be adequately addressed.
- The functionality of shared variables should be sufficient to permit implementation of commonly known algorithms involving shared memory.
- The definition of shared variables must be at least as precise as the rest of the language, and it should be possible to construct a correct formal model of shared variable semantics.
- Ada must be able to be implemented efficiently on parallel processor machines with multiple non-coherent caches.
- Ada programs must be able to control memory mapped input/output devices. If this cannot be achieved in full generality, then the extent to which Ada *can* address this problem must be well defined.
- Ada shall map efficiently onto shared memory multi-processor architectures.

Note that we did not include in the above list a requirement that shared variables be marked. It is hard to see how this requirement can be avoided. However, it is still a means to an end, rather than an end in itself.

References

1. Bryan, D. An Algebraic Specification of the Partial Orders Generated by Concurrent Ada Computations. Proceedings of Tri-Ada '89, Oct., 1989, pp. 225–241. ISBN: 0–89791–29–9, ACM Order Number: 825891.

2. Dewar, R. B. K., Flynn, S. Schonberg, E, and Shulman, N. Distributed Ada on Shared Memory Multiprocessors. Distributed Ada 89 Conference, 1990.

3. Flynn, S. F. *SMARTS — Shared-memory Multiprocessor Ada Run Time Supervisor.* Ph.D. Th., New York University, Courant Institute, 1988. E. Schonberg, advisor.

4. Dinning, A., Ichbiah, E., Kanony, P., Schonberg, E., and Schonberg, E. On-the-fly Detection of Access Anomalies in Ada Programs. New York University, Courant Institute, 1990. In preparation.

5. Shulman, N. V. *The Semantics of Shared Variables in Parallel Programming Languages.* Ph.D. Th., New York University, Courant Institute, 1987. Robert Dewar, advisor.

Appendix A: Section 9.11 of ANSI/MIL-STD-1815A, Shared Variables

1 The normal means of communicating values between tasks is by entry calls and accept statements.

2 If two tasks read or update a shared variable (that is, a variable accessible by both), then neither of them may assume anything about the order in which the other performs its operations, except at the points where they synchronize. Two tasks are synchronized at the start and at the end of their rendezvous. At the start and at the end of its activation, a task is synchronized with the task that causes this activation. A task that has completed its execution is synchronized with any other task.

3 For the actions performed by a program that uses shared variables, the following assumptions can always be made:

4 • If between two synchronization points of a task, this task reads a shared variable whose type is a scalar or access type, then the variable is not updated by any other task at any time between these two points.

5 • If between two synchronization points of a task, this task updates a shared variable whose type is a scalar or access type, then the variable is neither read nor updated by any other task at any time between these two points.

6 The execution of the program is erroneous if any of these assumptions is violated.

7 If a given task reads the value of a shared variable, the above assumptions allow an implementation to maintain local copies of the value (for example, in registers or in some other form of temporary storage); and for as long as the given task neither reaches a synchronization point nor updates the value of the shared variable, the above assumptions imply that, for the given task, reading a local copy is equivalent to reading the shared variable itself.

8 Similarly, if a given task updates the value of a shared variable, the above assumptions allow an implementation to maintain a local copy of the value, and to defer the effective store of the local copy into the shared variable until a synchronization point, provided that every further read or update of the variable by the given task is treated as a read or update of the local copy. On the other hand, an implementation is not allowed to introduce a store, unless this store would also be executed in the canonical order (see 11.6).

9 The pragma SHARED can be used to specify that every read or update of a variable is a synchronization point for that variable; that is, the above assumptions always hold for the given variable (but not necessarily for other variables). The form of this pragma is as follows:

pragma SHARED (*variable*_simple_name);

10 This pragma is allowed only for a variable declared by an object declaration and whose type is a scalar or access type; the variable declaration and the pragma must both occur (in this

order) immediately within the same declarative part or package specification; the pragma must appear before any occurrence of the name of the variable, other than in an address clause.

11 An implementation must restrict the objects for which the pragma SHARED is allowed to objects for which each of direct reading and direct updating is implemented as an indivisible operation.